

# Mecanismos de Troca de Mensagens

Prof. Gustavo Girão  
[girao@imd.ufrn.br](mailto:girao@imd.ufrn.br)

Baseado nos slides do professor Alexandre Carissimi (UFRGS)

# Roteiro

- Paradigmas de comunicação
  - Troca de mensagens
- Conceitos de redes de computadores
- Sockets
  - Interface
  - Criação
  - Inicialização
  - Requisitando
  - Aceitando
  - Fechando
- Exercícios

# Paradigmas de comunicação

- Variáveis compartilhadas
  - Memória de dados “visível” por todos
  - Comunicação implícita
  - Normalmente utilizada por threads
  - Problemas de condição de corrida
- Troca de mensagens
  - Normalmente utilizada em ambientes com memória distribuída
    - ✧ Não há compartilhamento de uma memória de dados
  - Utilizada por processos
  - Comunicação explícita
    - ✧ Pode tornar o código complexo

# Troca de mensagens

- Diversas implementações
  - Message Passage Interface (MPI)
    - ✧ Faz uso de uma biblioteca com funções de programação paralela e concorrente
  - Sockets
    - ✧ Solução de conexão TCP ou UDP entre aplicações que podem estar na mesma máquina ou em máquinas diferentes

# Troca de Mensagens

- A abordagem dos algoritmos de programação concorrente normalmente seguem o modelo cliente-servidor
  - Deriva-se do modelo mestre-escravo:
    - ✧ Um processo (mestre) dispara tarefas a serem realizadas por outros processos (escravos). Estes últimos retornam o resultado da computação para o mestre.
      - Execução finaliza após um determinado número de ciclos como este

# Sockets - Introdução

- Aplicações em ambientes de rede seguem dois modelos:
  - Cliente-servidor
  - Peer-to-peer
- Interface de programação (API) para ambientes de redes
  - Permite aplicações acessarem serviços de protocolos de redes
  - Conjunto de funções implementadas por uma biblioteca
  - Características desejáveis:
    - Genérica
    - Independente de sistema operacional
    - Suporte a comunicações orientadas a mensagens e a conexão
    - Prover funcionalidades equivalentes a camada de apresentação (MR-OSI)
      - Mascaram "formatos de dados" (inteiros, strings, *byte order* etc.)

# Redes de Computadores

- Conceito de endereço
  - Uma identificação única de um **interface** de rede

Vamos tentar abstrair a maior parte dos conceitos de redes de computadores envolvidos!

Isso será visto nas disciplinas de: Redes de Computadores e Programação Concorrente!

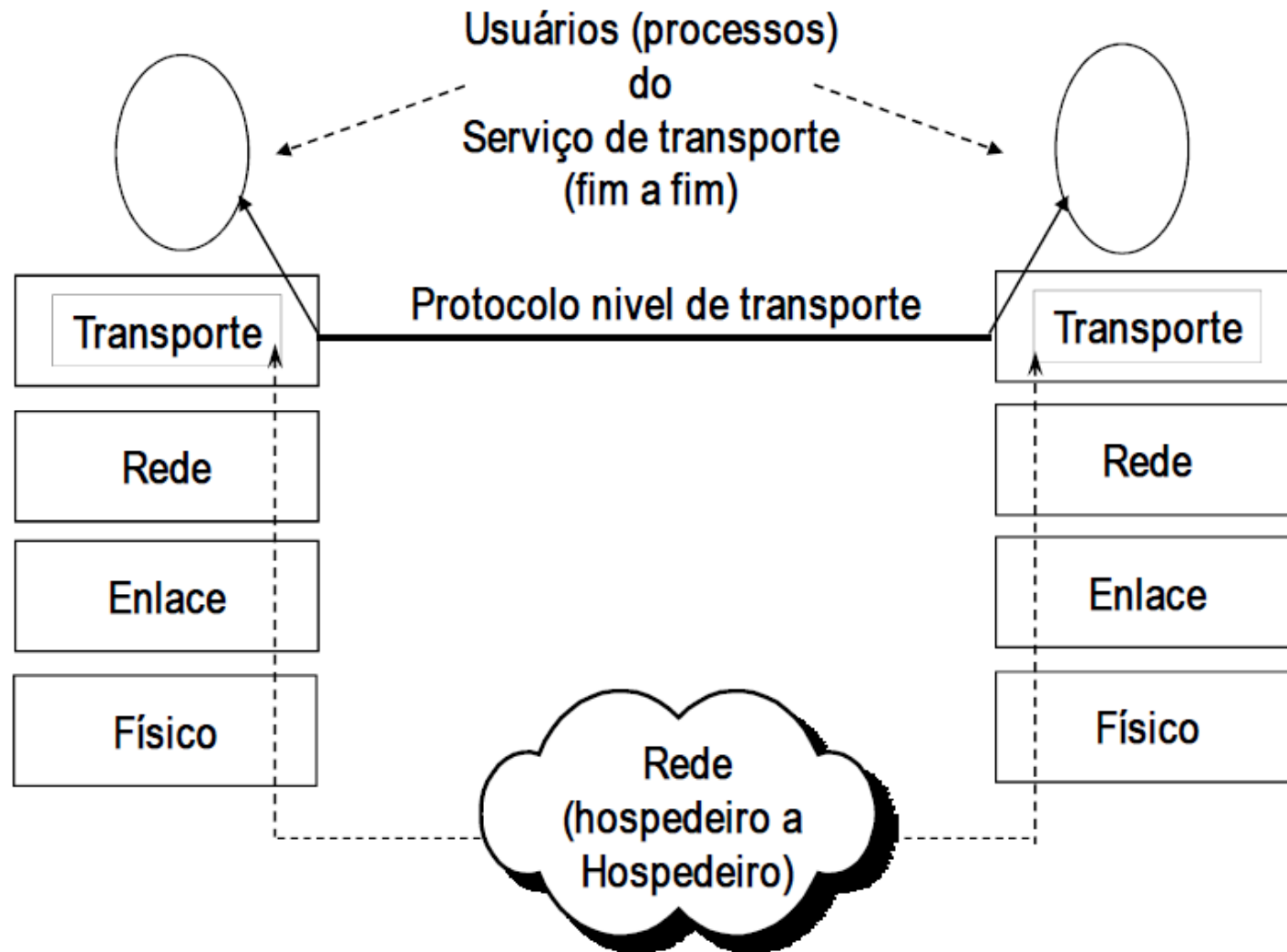
- Um mesmo computador pode ter diversas portas (visto que podem haver diversas aplicações executando)
- Uma conexão é caracterizada por endereço + porta + protocolo de comunicação

# Redes de Computadores

- Camada de rede na Internet
  - Endereço IP
    - Identificador de máquina
    - Embute informações de roteamento
- Camada de transporte na Internet
  - Comunicação fim a fim (processo a processo)
    - Processos são identificados por portas
  - Serviços:
    - Orientado a conexão (Transmission Control Protocol -TCP)
    - Não orientado a conexão (User Datagram Protocol – UDP)

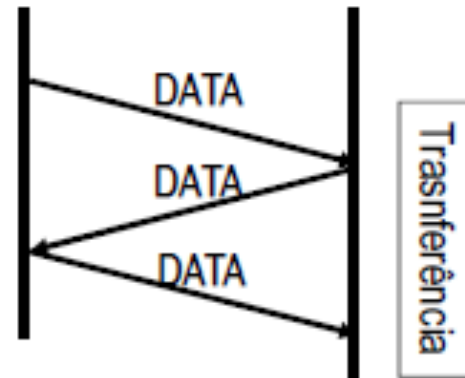


# Contexto dos protocolos



# User Datagram Protocol

- Não orientado a conexão
- Orientado a mensagem
  - Dados são delimitados na T-PDU
- Modelo de falhas:
  - Validade: falhas por omissão, isso é, não entrega
  - Integridade: pode haver perdas, erros de ordenamento e duplicação

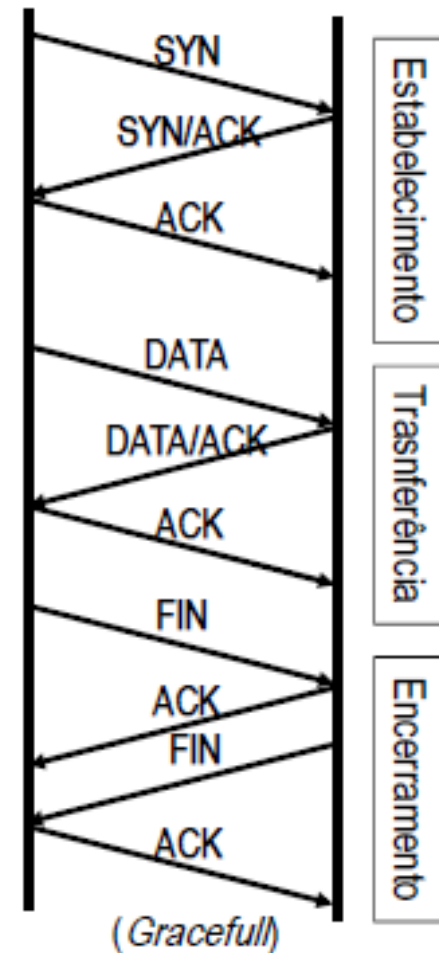


PDU = *Protocol Data Unit* (equivale a um “pacote” com cabeçalho e dados)

T = camada de transporte

# Transmission Control Protocol

- Orientado a conexão
  - Comunicação bidirecional
- Noção de *byte stream* (fluxo de dados)
  - Dados não são delimitados pela T-PDU
- Conexão fornece garantia de entrega, não duplicação e ordem
  - Não é uma comunicação 100% confiável, pois conexões podem ser rompidas ou desfeitas
    - Falha de rede ou do processo?
- Modelo falhas:
  - Validade: garante a entrega das mensagens
  - Integridade: garante ordenamento e não duplicação



# Conceito de Porta

Porta	Protocolo	Aplicação
20	TCP	FTP-data
21	TCP	FTP-control
25	TCP	SMTP
53	TCP/UDP	DNS
80	TCP	HTTP
110	TCP	POP3
161	UDP	SNMP

<div></div>	<div></div>	<div></div>	<div></div>
TCP		UDP	
IP Internet Protocol			
Interface de rede			

- Número de 16 bits usado como identificador
- Dois tipos de portas
  - Bem conhecidas (*well know ports*): 1 a 1023
  - Efêmeras (*ephemeral ports*)
    - *Registered ports*: 1024 a 49151
    - *Dynamics and/or private ports*: 49152 a 65535
- Uma aplicação é completamente identificada por:
  - End. IP + Porta + Protocolo (TCP ou UDP)
  - Uma porta não pode ser compartilhada por vários processos (exceção: multicast IP)
  - Portas TCP são independentes de Portas UDP
    - Porta 100 (TCP) ≠ Porta 100 (UDP), mas se convencionou "alocar" as duas simultaneamente para um mesmo protocolo

# Modelo Cliente-Servidor

- Paradigma muito usado na implementação de aplicações em rede
  - Pode ser usado com serviços orientados a conexão ou não
- Modelo assimétrico
  - Cliente envia requisições para o servidor
  - Servidor envia respostas para o cliente
- Cliente
  - Usuário de um serviço provido por um processo (servidor)
  - Solicita execução do serviço no servidor e espera uma resposta
    - Em serviços orientados a conexão é o cliente que solicita a abertura da conexão para enviar requisições ao servidor
- Servidor
  - Fornece um serviço (aplicação) a outros processos (clientes)
  - Espera passivamente a solicitação de clientes
- Necessário identificar:
  - Máquina: endereço IP
  - Processo: Porta
  - Protocolo: TCP ou UDP

# Sockets

- Abstração de um ponto de comunicação (*endpoint*)
- Prove suporte a vários tipos de protocolos (família)
  - Noção de ponto de comunicação independente de protocolo
- API *socket* é genérica contendo:
  - Suporte para várias famílias de protocolos
  - Funções para criar, nomear, conectar e destruir *sockets*
  - Esperar pedido de conexão e terminar
  - Funções para envio e recepção
  - Configuração, gerenciamento e controle de *sockets*
  - Tratamento de erros

# Sockets no UNIX

- Forma de comunicação bidirecional
- Dois domínios básicos:
  - Protocolos de transporte da Internet (AF\_INET)
    - Canal de comunicação é definido pela associação (5-tupla)
      - [IP\_destino, Porta\_destino, IP\_fonte, Porta\_fonte, Protocolo]
      - Envolve um par de sockets (*endpoint* local e *endpoint* remoto)
  - UNIX (AF\_UNIX)
    - Permite comunicação com interface sockets em um mesmo host
    - Outro mecanismo de IPC local
    - Vantagem é o desempenho (se comparado com AF\_INET)
    - Canal de comunicação é um “nome” no sistema de arquivos



# Interface de Sockets

- *Socket* é um descritor de arquivo
  - Paradigma *abrir-ler-escrever-fechar*
- As primitivas básicas são:
  - `socket()`
  - `bind()`
  - `listen()`
  - `accept()`
  - `connect()`
  - `write()`, `sendto()`
  - `read()`, `recvfrom()`
  - `close()`



# Principais funções da API

<b>socket</b>	Cria um novo descritor para comunicação
<b>connect</b>	Iniciar conexão com servidor
<b>write</b>	Escreve dados em uma conexão
<b>read</b>	Lê dados de uma conexão
<b>close</b>	Fecha a conexão
<b>bind</b>	Atribui um endereço IP e uma porta a um socket
<b>listen</b>	Coloca o socket em modo passivo, para “escutar” portas
<b>accept</b>	Bloqueia o servidor até chegada de requisição de conexão
<b>recvfrom</b>	Recebe um datagrama e guarda o endereço do emissor
<b>sendto</b>	Envia um datagrama especificando o endereço

# Criação

```
int socket(int domain, int type, int protocol);
```

- *Domain* especifica família de endereçamento (AF\_INET ou AF\_UNIX)
- *Type* (para AF\_INET)
  - Não orientado a conexão (UDP): SOCK\_DGRAM
  - Orientado a conexão (TCP): SOCK\_STREAM
- *Protocol*: protocolo a ser usado para aquele tipo de serviço
  - Casos exista mais de um protocolo oferecendo o mesmo tipo de serviço
  - O valor 0 indica *default*
- Retorna descritor ou -1 em caso de erro

# Inicialização

- Necessário atribuir um nome para disponibilizá-lo às aplicações
  - AF\_UNIX: definir um nome em um *path*
  - AF\_INET: determinar porta e IP
- “Batizado” através da chamada *bind()*

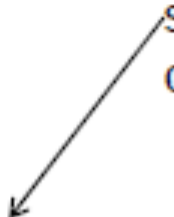
```
int bind(  
    int socket,                /* descritor do socket a ser inicializado */  
    const struct sockaddr *address, /* ponteiro para estrutura com valores inicialização */  
    size_t address_len         /* tamanho da estrutura de inicialização */  
);
```

- Estrutura de inicialização: *sockaddr\_un* ou *sockaddr\_in*

# Endereço de um socket para AF\_INET

- Necessário posicionar:
  - Endereços IP (32 bits para o IPv4)
  - Endereço porta (16 bits)
- Usado um tipo de dados especial (*struct sockaddr\_in*)

```
struct sockaddr_in {  
    uint_t          sin_len;  
    sa_family_t     sin_family; /* AF_INET */  
    in_port_t       sin_port;   /* Port number */  
    struct in_addr   sin_addr;   /* Internet address */  
    char            sin_zero[8];  
};  
  
struct in_addr {  
    in_addr_t       s_addr;  
};
```



# Fila de requisições

- Necessário armazenar pedidos de aberturas de conexão até que eles possam ser atendidos
  - Implica em ter uma fila (e um tamanho) desses pedidos
    - Em caso de overflow, os pedidos de conexão são perdidos
  - Apenas para serviços orientados a conexão (SOCK\_STREAM)
- Chamada *listen()*

```
int listen(  
    int socket,    /*descritor do socket */  
    int backlog    /* tamanho da fila */  
);
```

# Aceitando conexões: servidor

- Apenas para serviços orientados a conexões
- Recupera o primeiro pedido da fila de requisições (*listen*)
  - Bloqueia se a fila estiver vazia
    - Procedimento *default*, mas é possível modificar com a opção `O_NONBLOCK` passada via chamada *fcntl()*
- Chamada *accept()* executada pelo servidor

```
int accept(  
    int socket,                /* descritor do socket onde se espera pedidos */  
    const struct sockaddr *address, /* ponteiro estrutura que recebera dados do cliente */  
    size_t address_len         /* tamanho da estrutura de dados do cliente */  
);
```

- Importante: retorna o descritor de um novo *socket*
  - O que será usado para comunicação com o cliente

# Requisitando conexões: Cliente

- Apenas para serviços orientados a conexões
- Executado pelo cliente para solicitar abertura de conexão
- Chamada `connect( )` executado pelo cliente

```
int connect(  
    int socket,                /* descritor do socket do cliente – não inicializado */  
    const struct sockaddr *address, /* ponteiro estrutura de dados do servidor */  
    size_t address_len         /* tamanho da estrutura de dados do servidor */  
);
```

- *Default* é bloquear até a conexão ser aceita ou expirar *timeout*
  - Comportamento pode ser modificado via `fcntl( )` com opção `O_NONBLOCK`

# Encerrando um socket

- Terminar uma conexão entre um cliente e um servidor
- Chamada *close()*
  - Similar ao que se faz com um arquivo
  - Deve-se chamar o encerramento em ambos lados (cliente e servidor) para evitar descritores
- O encerramento pode não ser imediato se ainda houver dados em trânsito (encerramento *graceful*)



# Funções de conversão

- Conversão de endereços IP (ASCII) em formato de rede (binário)
- Funções para IPv4
  - `inet_aton`: end. IPv4 (ASCII) para binário
  - `inet_ntoa`: binário para end. IPv4 (ASCII)
  - `inet_addr` (obsoleta): similar a `inet_aton`
- Funções para IPv4 e IPv6
  - `inet_pton`: presentation to network
  - `inet_ntop`: network to presentation

Necessário indicar a família (`AF_INET` ou `AF_INET6`)

# Comunicação por UDP

- Baseado em uma unidade de transmissão: datagrama
  - Tamanho limitado
- Transmitido do remetente ao destino sem confirmações ou tentativas de reenvio
  - Mensagens podem não chegar
- Aspectos a serem considerados
  - Tamanho da mensagem:
    - Erro no envio ou descarte na transmissão
  - Bloqueio: *send* é não bloqueante e o *receive* é bloqueante
  - Time-out: configurável via *sockoption*
  - Recepção anônima (INADDR\_ANY)

# Sockets sem Conexão (C)

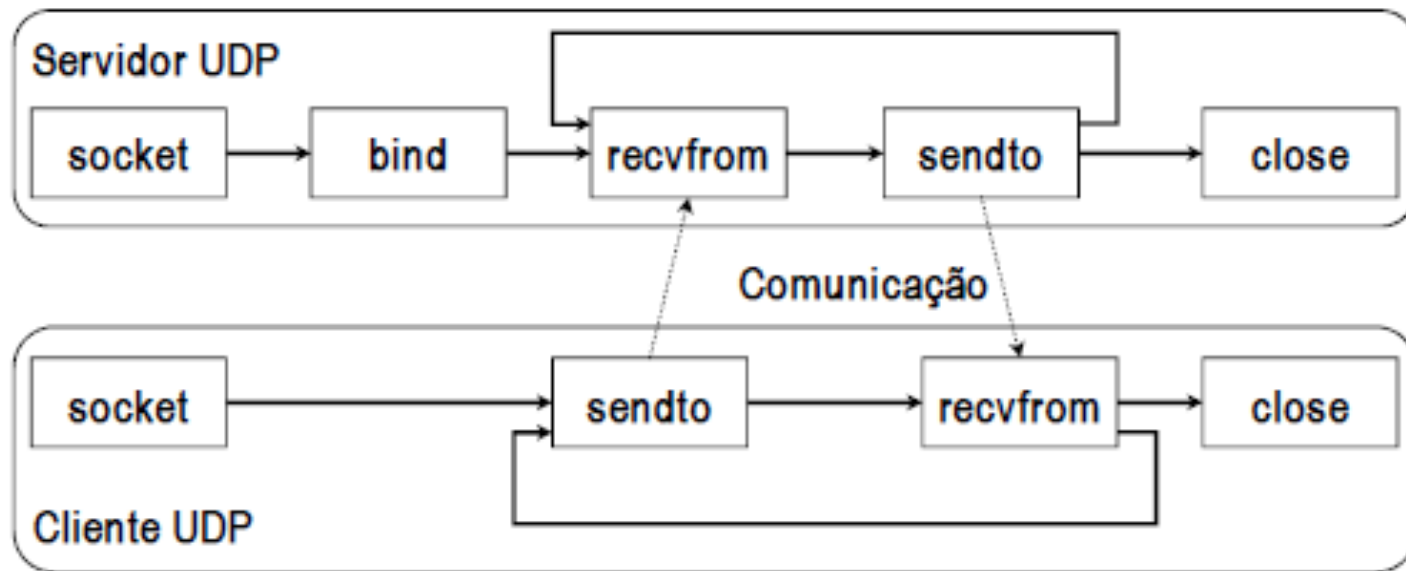
- Cliente:

- `s = socket(AF_INET, SOCK_DGRAM, 0);`
  - `sendto(s, msg, length, flags, destaddr, addrlen);`
  - `recvfrom(s, msg, length, flags, fromaddr, addrlen);`

- Servidor:

- `s = socket(AF_INET, SOCK_DGRAM, 0);`
  - `bind(s, dest, sizeof(dest));`
  - `recvfrom(s, msg, length, flags, fromaddr, addrlen);`
  - `sendto(s, msg, length, flags, destaddr, addrlen);`

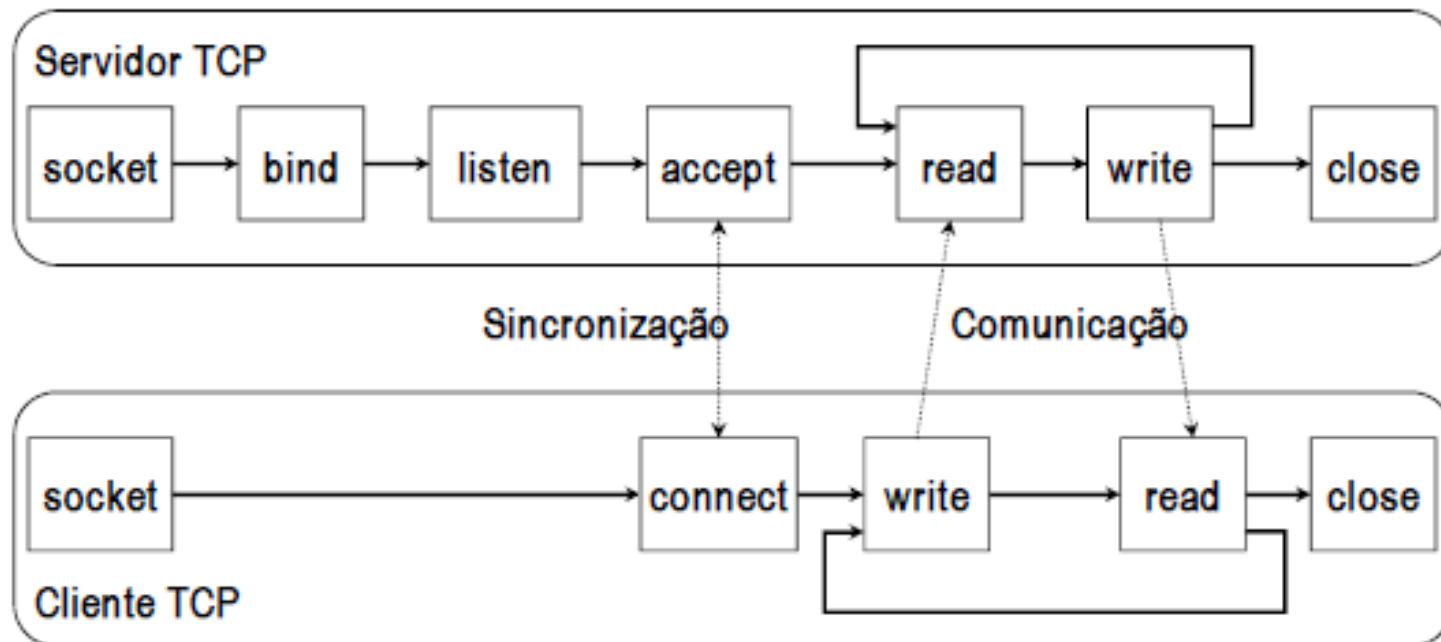
# Cientes e Servidores UDP



# Comunicação por TCP

- Define a abstração de fluxo (*stream*)
  - Dados podem ser lidos (*receive*) e escritos (*send*) na base de uma relação produtor-consumidor
- Características de um fluxo TCP
  - Não define limites de tamanho para as mensagens (tamanho variável)
  - Efetua controle de erro (evita perdas por erro)
  - Faz controle de fluxo (regula a taxa de leitura e escrita em fluxo para evitar perdas por overflow)
  - Garante entrega ordenada e a não-duplicação dos dados
  - Define a abstração de conexão como identificadores de processos e máquinas origem e destino

# Cientes e servidores TCP



# Exercício

- Crie dois programas em C:
  - Um chamado de **mestre** que envia a dois **escravos** dois vetores de inteiros com 10 elementos cada e em seguida aguarda um valor inteiro de cada **escravo** e os imprime na tela.
  - Um chamado de **escravo** que recebe do **mestre** o vetor de inteiros de 10 elementos e calcula a soma de todos estes elementos. Em seguida, envia de volta ao **mestre** o valor resultante.
- Utilize sockets com protocolo UDP
- Lembre-se de que o programa mestre precisa ter seu socket criado e caracterizado (bind) para que os escravos possam se conectar a ele
- Utilize os exemplos disponíveis no SIGAA

# Referências

- OLIVEIRA, Rômulo Silva de; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. **Sistemas operacionais**. 4. ed. Porto Alegre: Bookman, 2010. ISBN: 9788577805211.
  - **Capítulo 3**
- TANENBAUM, Andrew S.. **Sistemas operacionais modernos**. 3. ed. São Paulo: Prentice Hall, 2009. 653 p. ISBN: 9788576052371.
  - **Capítulo 2, Seção 2.3**
- SILBERCHATZ, A.; Galvin, P.; Gagne, G.; **Fundamentos de Sistemas Operacionais**, LTC, 2015. ISBN: 9788521629399
  - **Capítulo 6**



# Próxima Aula

- Exercícios com Sockets e MPI